ISA Transactions 118 (2021) 116–132

Contents lists available at ScienceDirect

ISA Transactions

journal homepage: www.elsevier.com/locate/isatrans

Practice article

Didactic platform for DC motor speed and position control in Z-plane

Tomàs Pallejà Cabré *, Albert Saiz Vela, Marcel Tresanchez Ribes, Javier Moreno Blanc, Jose Ribó Pablo, Francisco Clariá Sancho

University of Lleida, Department of Computer Science and Industrial Engineering, C\Jaume II, 69, 25001, Lleida, Spain

ARTICLE INFO

Article history:
Received 22 November 2019
Received in revised form 12 January 2021
Accepted 12 February 2021
Available online 22 February 2021

Keywords:
Control
Arduino
DC motor
PID
Z plane

ABSTRACT

This paper describes how to implement a low-cost didactic platform designed to teach or reinforce discrete control theory concepts. The controllers used in this work (P, PI, PD, and PID) are suitable for undergraduate students but the same platform could be used to explain and test advanced controllers to graduate students. This document shows, step by step, how to control a DC motor speed and position, along with the most common problems and its solutions, commonly overlooked in the literature. It also explains how to simulate the system behavior and compares the simulations with the real data, showing an average correlation coefficient of ρ=0.983.

1. Introduction

Teaching and leaning discrete control systems could be a challenging task. It involves multiple disciplines, such as electronics, physics, programming and theoretical concepts that use many branches of mathematics [1].

Having a hands-on control system platform allows students to put in practice the control theory foundations and improves the overall learning experience [2,3]. Commercial platforms can be very expensive [4] and usually, schools need to buy more than one to practice in small groups of students. In the literature there are three commercial platforms alternatives: (1) using simulators, like Simulink or LabVIEW [5–9], which are excellent tools to teach and test control systems but they lack on a real system hands-on experience. (2) using real remote laboratories [10,11], which are great but need a schedule for the students to access. And (3) creating a custom control teaching laboratory using low-cost hardware [12], which may have some issues related to the limited low-cost hardware performance. Thanks to the 3D printers and the Arduino microcontrollers, it is quite fast, easy and cheap to create a custom didactic control platform. For beginners, applying the theoretical concepts to a real system is not a trivial task and they often need some references. A good starting point is searching in the literature where it is easy to find information about the PID controllers. These are by far the simplest and yet most efficient solution to many real-world control problems [13–15]. A PID controller can be tuned using the famous Ziegler and Nichols method [16], the magnitude optimum criterion [17] and other techniques summarized in [18], but most of them are too advanced for beginners.

From an educational point of view, many papers show how to solve a specific control task, [19–22], but most of them assume or ignore small details that make it difficult to replicate for beginners, such as wiring, coding, step by step calculations or sampling methods among others. There are also papers very similar to this one [23,24] but focused on the academic results.

The proposed low-cost didactic platform has a 3D printed plastic frame that holds a 12 V DC motor, a microcontroller and a motor driver. With this basic setup, students have to control the motor speed and position and evaluate its performance in front of different scenarios. This manuscript was written to summarize all the steps and problems students had to overcome in the last four courses of the System Integration subject while implementing the digital controllers. These students are in the 4th year of the Degree in Automation and Industrial Electronic Engineering, and they already know electronics, differential equations, Fourier, Laplace and the Z transform. In this subject, they put in practice all this knowledge facing off real issues, and learning Simulink, C and C# coding.

We have three main goals to accomplish in this work, (1) to present a low-cost control teaching platform, (2) to create a discrete control theory guide for undergraduate students and professors that summarizes all the steps and problems that one could encounter while trying to control a DC motor for the first time and (3) to show the matching between the real motor performance, the simulations, and the analytical results. This paper does not pretend to teach analytical techniques for PID tuning, instead, it uses the MATLAB PID tuner application [25] for that purpose.

* Corresponding author.
E-mail addresses: tomas.palleja@udl.cat (T.P. Cabré), albert.saiz@udl.cat (A.S. Vela), marcel.tresanchez@udl.cat (M.T. Ribes), javier.moreno@udl.cat (J.M. Blanc), jose.ribo@udl.cat (J.R. Pablo), francisco.claria@udl.cat (F.C. Sancho).

https://doi.org/10.1016/j.isatra.2021.02.020

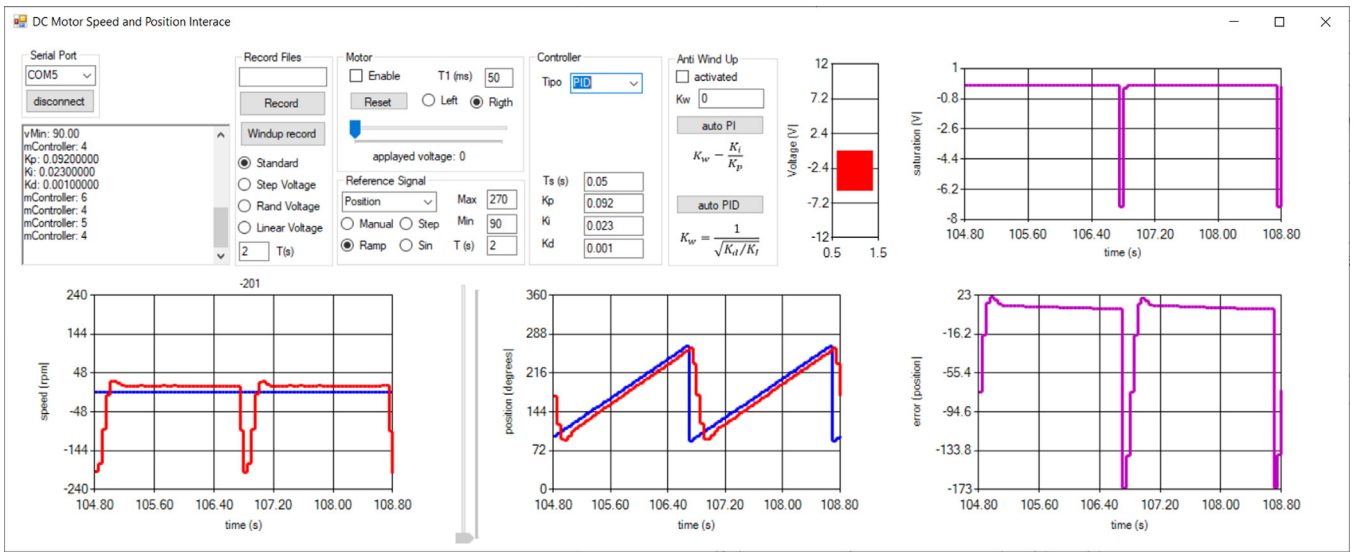

Fig. 1. Implemented system interface.

This paper is organized as follows. Section 2 describes the proposed platform and explains how to model a motor transfer function. Section 3 shows the speed and position control diagram and its digital open-loop transfer functions. Section 4 shows how to transform a controller from the Laplace transfer function to an equation in differences, ready to code in the microcontroller. Section 5 exposes the nonlinearities and how to deal with them, both in *Simulink* and the microcontroller. Finally, Section 6 presents some experiments to verify the controllers' performance and the comparison between the real and the simulated output, both for speed and position control.

2. Platform description

This section defines the materials used in this work and the methods to obtain the motor mathematical model to properly design a closed-loop discrete control system.

2.1. Material

Fig. 2 shows the electronic schematic overview while Fig. 3 illustrates the schematic front view of the 3D printed framework. This structure has 360 angular degree marks and holds a DC motor connected to an arrow (case a) or a pulley (case b).

Also, there is a variable weight which can be detached anytime to simplify the analysis. When the motor spins the weight turns around (case a) or up and down (case b), acting as a variable or constant perturbation respectively.

Undergraduate students (4th course) of the Automation and Industrial Electronic Engineering Degree invested 28 h in developing a system interface (Fig. 1). It allows them to connect the serial port; plot and record the data from Arduino; set the controllers; create different reference signals and, also, acts as a *HyperTerminal* for debugging. Table 1 summarizes the hardware and software relevant information.

An Arduino Due [26] development board was used. It is an open-source electronics platform based on easy-to-use hardware and software for building electronic projects. Arduino consists of two main parts: a physical programmable board, and an Integrated Development Environment (IDE). This is used to write and upload the code to the microprocessor. The microcontroller PWM pins (max 30 mA) cannot supply enough current to run the motor (from 29 to 300 mA) then, a motor driver (H-Bridge) is

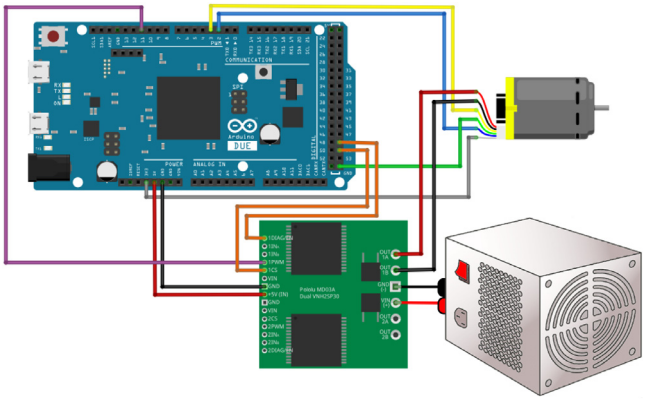

Fig. 2. Power supply unit, Arduino Due, motor, H bridge, encoder, and motor schematic connections.

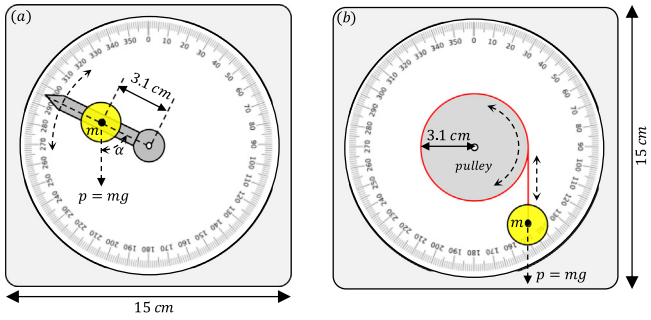

Fig. 3. (A) Arrow and detachable weight, acting as a variable perturbation. (B) Pulley and weight, acting as a constant perturbation.

needed [27]. The voltage applied to the motors was controlled by a PWM signal from the Arduino. Additionally, *enable* and *direction* digital pins were used (see Fig. 2).

The motor [28] is a medium-power gear motor. It has a biphasic encoder giving 1632.672 pulses per revolution (see Table 1 and Eq. (1)), and an angular resolution of 0.2205 degrees per pulse (Eq. (2)).

$$PPR = 34.014 \cdot 24 \cdot 2 = 1632.672 \frac{\text{pulses}}{\text{rev}} \tag{1}$$

Table 1
Platform specifications.

	Interface	Microcontroller
Device	PC, i5, Windows 10	Arduino DUE
IDE	MS visual studio	Arduino software
Code	C#	C++
Serial baud rate	250 000 bauds	250 000 bauds
DAC	–	12 bits
Reference signals	Manual, step, ramp, sinus, and random	–
	H-Bridge	
Brand/model	Pololu dual MC33926	
Motors	2 (only one is used in this work)	
Voltage	-8 V	
Current	3 Amps per motor	
Control signal	PWM	
	Motor	
Brand	Pololu	
Nominal voltage	12 V	
Reduction gears	34.014:1	
Stall current	2.1 A (at 12 V)	
Maximum speed	230 rpm (12 V)	
Minimum speed	10 rpm (0.7 V, no load)	
Maximum torque	0.44 N m	
	Motor encoder	
Type	Hall-effect	
Hall sensors	2	
Magnet poles	24	

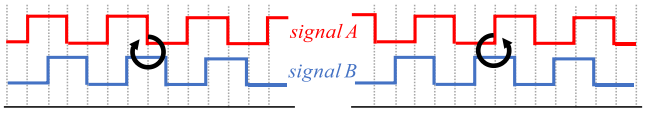

Fig. 4. Clockwise and counterclockwise biphasic encoder signals.

$$\frac{360\ \text{deg}}{1\ \text{rev}} \cdot \frac{1\ \text{rev}}{1632.672 \cdot \text{pulses}} = 0.2205 \frac{\text{deg}}{\text{pulse}} \tag{2}$$

The motor position and speed were calculated using the biphasic encoder signals, connected to two Arduino digital pins (*PinA* and *PinB*). Since the encoder use open-drain MOSFETs, it is very important to set those pins with a pull-up resistor (INPUT_PULLUP).

Fig. 4 shows that when the motor is going clockwise the encoder signal A falls while signal B is high. On the other hand, when the motor is going counterclockwise, signal A falls while signal B is low. Appendix A shows the functions called each time signals A or B fall or rise (*funA* and *funB*). Those functions have the logic to update the number of pulses (*np*) according to the motor spinning direction. The variables *signalA*, *signalB* and *np* must be decelerated as volatile because they are updated by an external interruption.

The microcontroller function which estimates both speed and position (*funTimer*) was called by a timer interrupt every T_1 seconds. To estimate the motor position (Eq. (3)) the degrees/pulses relationship (Eq. (2)) times the number of actual pulses was used. The motor instant speed was computed as the variation of pulses (Δnp) in a small period of time (Δt), where Δt is the period T_1. The Δnp is computed as the np variation between consecutive samples (Eq. (4)). Finally Eq. (5) was used to estimate the motor's current speed.

$$\underset{[\text{deg}]}{position} = np\ \text{pulses} \cdot 0.2205 \frac{\text{deg}}{\text{pulse}} \tag{3}$$

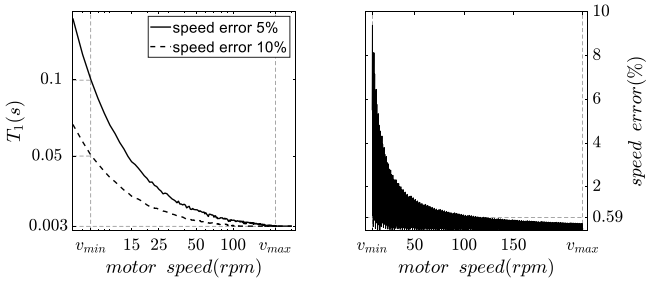

Fig. 5. Speed period estimation.

$$\begin{aligned} \Delta t &= T_1 \\ \Delta np &= np\,(k) - np\,(k-1) \end{aligned} \tag{4}$$

$$\underset{[\text{rpm}]}{speed} = \frac{\Delta np\ \text{pulses}}{\Delta t\ \text{s}} \cdot \frac{1\ \text{rev}}{1632.672\ \text{pulses}} \cdot \frac{60\ \text{s}}{1\ \text{min}} = \frac{0.03675 \cdot \Delta np}{T_1} \tag{5}$$

A key point is to select a convenient T_1, it has to be small enough to have a reasonable updating frequency but not too short that Δnp is excessively small (increasing the speed error by rounding effects). The speed error was estimated as a function of the motor speed (φ [rpm]) and the sampling period (T_1). The analytic number of pulses between sampling periods (Δanp) is more likely to be a $\mathbb{R}$ number (Eq. (6)) while the real number of pulses will always be a $\mathbb{N}$ number. By rounding the Δanp to the nearest integer towards zero and using Eq. (5) (to transform from pulses to speed), it is possible to estimate the real speed error (Eq. (7)).

Fig. 5, left, shows the motor speed and T_1 relationship to have an speed error of 5 and 10%, between v_{min} (7 rpm) and v_{max} (220 rpm). For $T_1 \sim 50$ ms it creates an absolute error of 10% at v_{min}. Fig. 5, right, shows that for $T_1 = 50$ ms, this error quickly decreases while the motor speed increases, having an average error of ~0.6%. Finally, T_1 was set at 50 ms.

$$\Delta anp = PPR \frac{\text{pulses}}{\text{rev}} \cdot \varphi \frac{\text{rev}}{\text{min}} \cdot \frac{1\ \text{min}}{60\ \text{s}} \cdot T_1 s = \frac{PPR \cdot \varphi \cdot T_1}{60}\ \text{pulses} \tag{6}$$

$$SpeedError\ (\%) = \frac{100\left(\varphi - \frac{0.03675 \cdot \lfloor \Delta \text{anp} \rfloor}{T_1}\right)}{\varphi} \tag{7}$$

2.2. System identification and modeling

To obtain the plant (motor) transfer function ($G_p(s)$), it is necessary to derive its time-domain mathematical model (differential equations) and then perform the Laplace transform. Fig. 6 shows the electrical equivalent circuit of a DC motor [29] and Table 2 indicates its physical parameters and the signals involved.

The torque generated by the motor (Eq. (8)) is proportional to the current, and the back electromotive force is proportional to the shaft angular speed $\dot{\theta}(t)$ (Eq. (9)). Then, applying Newton's 2nd law and Kirchhoff's voltage law in Fig. 6 results in Eqs. (10) and (11) respectively.

$$T(t) = K_t i(t)\ [\text{N m}] \tag{8}$$

$$\varepsilon(t) = K_e \frac{d\theta(t)}{dt}\ [\text{V}] \tag{9}$$

$$J \frac{d^2\theta(t)}{dt^2} = T(t) - b \frac{d\theta(t)}{dt} \tag{10}$$

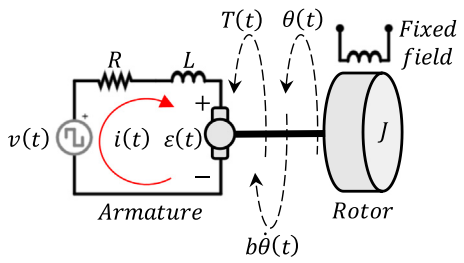

Fig. 6. Electrical equivalent circuit of a DC motor.

Table 2
Motor physical parameters.

Symbol	Description	Units
$\theta\,(t)$	Shaft angular position	rad
$T\,(t)$	Torque	N m
$v\,(t)$	Input voltage	V
$i\,(t)$	Current	A
$\varepsilon\,(t)$	Electromotive force	V
R	Equivalent resistance	Ω
L	Equivalent inductance	H
J	Moment of inertia	kg m^2
b	Motor viscous friction constant	N m s
K_e	Electromotive force constant	V/rad/s
K_t	Motor torque constant	N m/A

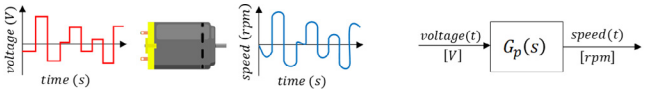

Fig. 7. Real (left) and modeled (right) open-loop system identification schematics.

$$v\,(t) - Ri\,(t) - L\frac{di\,(t)}{dt} - \varepsilon\,(t) = 0 \tag{11}$$

The Laplace transform of Eq. (10) and (11) results in Eqs. (12) and (13). Replacing $I\,(s)$ from Eqs. (12) to (13) results in the motor transfer function (Eq. (14)), which relates the input voltage with the shaft angular position. It is usually more interesting to have the relationship between the input voltage and the shaft speed, in this case, it is as simple as multiplying both terms of Eq. (14) by the complex variable s, obtaining the desired transfer function (Eq. (15)). Remember that $s\Theta\,(s) = \dot{\Theta}\,(s)$.

$$Js^2\Theta\,(s) = K_t I\,(s) - bs\Theta\,(s) \tag{12}$$

$$V\,(s) - RI\,(s) - sLI\,(s) - k_e s\Theta\,(s) = 0 \tag{13}$$

$$\frac{\Theta\,(s)}{V\,(s)} = \frac{K_t}{s\left[(Js+b)\,(Ls+R) + K_eK_t\right]}\left[\frac{\text{rad}}{\text{V}}\right] \tag{14}$$

$$G_p\,(s) = \frac{\dot{\Theta}\,(s)}{V\,(s)} = \frac{K_t}{(Js+b)\,(Ls+R) + K_eK_t}\left[\frac{\text{rad/s}}{\text{V}}\right] \tag{15}$$

If the motor manufacturer provides those physical parameters, the transfer function is easily obtained from Eq. (15). If not, it can be identified by applying a known voltage $v\,(t)$ to the input and analyzing the speed response (Fig. 7). The motor model was obtained loading this recorded data in the MATLAB *System Identification ToolBox* and specifying a second-order model without zeros (Eq. (15)).

To do so, three input signals were generated, a 6 V step, 20 random steps (from −12 to 12 V) of two seconds each, and a 40 s ramp ($v = 0.6t - 12$). Table 3 shows the obtained transfer functions and their fitting accuracy.

With those transfer functions and their corresponding inputs, the output signals were calculated (Figs. 8–10). In every case the visual results are satisfactory, also supported by their accurate

Table 3
System identification.

Input	Transfer functions, $G_p\,(s)$	Fitting
Step (6 V)	$G_p^{step}\,(s) = \dfrac{18\ 150}{s^2 + 54.99\ s + 927.1}$	99.11%
Rand (±12 V)	$G_p^{rand}\,(s) = \dfrac{16\ 070}{s^2 + 50.61\ s + 859.7}$	95.57%
Ramp (±12 V)	$G_p^{ramp}\,(s) = \dfrac{2328}{s^2 + 15.61\ s + 126.1}$	96.53%

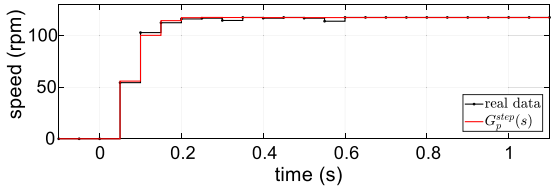

Fig. 8. Plant 6 V step response and its estimated response using $G_p^{step}\,(s)$.

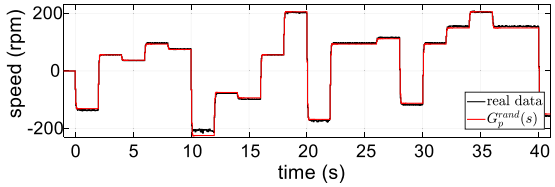

Fig. 9. Plant random steps response and its estimated response $G_p^{rand}\,(s)$.

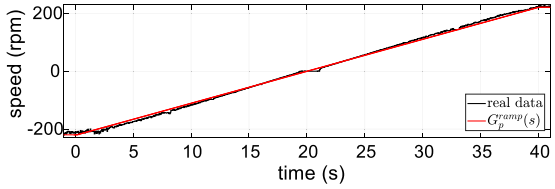

Fig. 10. Plant ramp response and its estimated response using $G_p^{ramp}\,(s)$.

fitting ratios. The $G_p^{step}\,(s)$ and $G_p^{rand}\,(s)$ have very similar transfer functions. The $G_p^{ramp}\,(s)$ has a slightly better fitting ratio than $G_p^{rand}\,(s)$, this is because $G_p^{ramp}\,(s)$ has no abrupt signal changes (bandwidth of 30 rad/s). On the other hand, due to the multiple steps, the $G_p^{rand}\,(s)$ output signal has the largest bandwidth (50 rad/s). For this reason, although it has the worst fitting ratio, $G_p^{rand}\,(s)$ was selected as the motor plant to be used.

3. Control system blocks diagram

This section contains the main point of the work. It is crucial to fully understand the whole picture to do the math properly. Figs. 11 and 12 show the diagrams to control the motor speed while Figs. 13 and 14 show the diagrams to control the motor position. The block *int. pulse register* (see Figs. 11 and 13) contains the interrupt functions (see Appendix A) to transform from encoder pulses to *np* and the block *var* symbolizes all the variable present in the diagram. The reference signal ($v_r\,(t)$ for speed control and $pos_r\,(t)$ for position control) was generated in the computer (using the system interface) and sent to the Arduino through the serial port every T_3 s. Arduino receives and holds this reference (using the serial event interruption) and answers back every T_2 s with a detailed report including: $t\,(k)$, $e\,(k)$, $u\,(k)$, $v\,(k)$, $pos\,(k)$ and $v_r\,(k)$ or $pos_r\,(k)$ depending on which is the controlled variable. The interface (computer) receives this report by another serial event interruption and plots/save all this data.

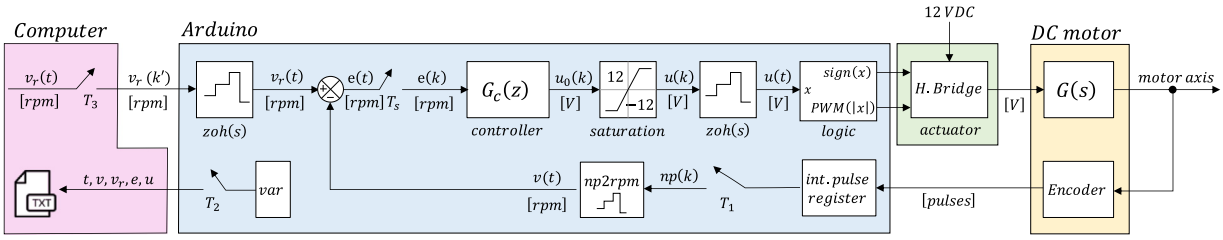

Fig. 11. Full speed block diagram.

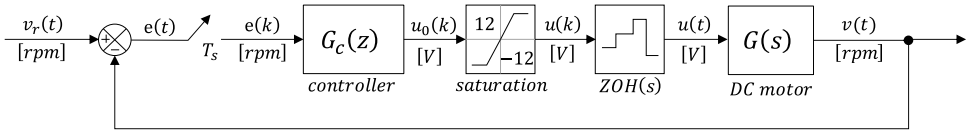

Fig. 12. Simplified speed block diagram.

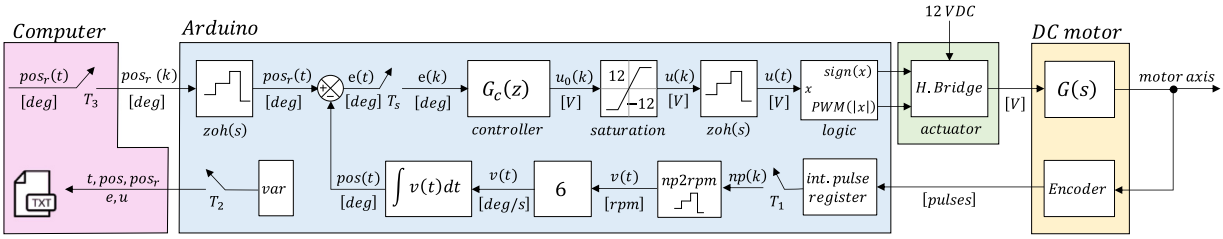

Fig. 13. Full position block diagram.

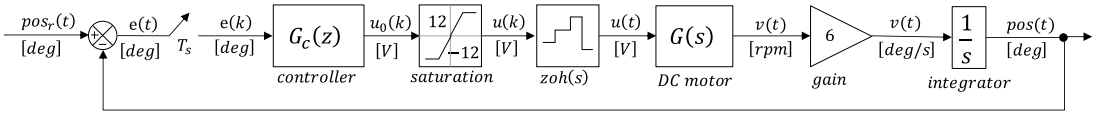

Fig. 14. Simplified position block diagram.

3.1. Timers

In this work there are four timers involved, with periods: T_s, T_1, T_2 and T_3 (see Figs. 11 and 13). The period T_1 was already established at 50 ms in Section 2. To work with discrete time control systems, it is reasonable to synchronize all samplers at the same frequency, for this reason T_1, T_2 and T_3 were also set at 50 ms. It is important to highlight that T_3 (interface) is not synchronized with the others (Arduino).

Following this logic, T_s should be set at 50 ms as well, but first, it has to be confirmed that Shannon's theorem [30] applies.

It says that the sampling frequency ($\omega_s = 2\pi/T_s$) must be, at least, twice the highest-frequency (ω_m) component present in the continuous-time signal.

An open-loop continuous-time output signal was simulated using the $G_p^{rand}(s)$ transfer function and 30 random steps as the input signal (from −12 to 12 V, of 2 s each). Appendix B details the MATLAB code to create Fig. 15, which shows the output signal module spectrum and its 2 first complementary components (red and blue) at $\pm[\omega_s, 2\omega_s]$.

For $T_s = 50$ ms and $\omega_m = 50$ rad/s, it is possible to estimate ω_s and prove Shannon's theorem (Eq. (16)). It is also possible to estimate the number of samples per cycle (Eq. (17)), which according to [31], is recommended to be between 8 and 10. To estimate ω_d, the $G_p^{rand}(s)$ characteristic equation ($s^2 + 50.61s + 859.7 = 0$) and the relationships shown in Eq. (18) were used. As a result, ω_d is 14.81 rad/s and the number of samples per cycle is 8.485. To see it graphically, Fig. 16 shows the $z\{ZOH(s)\,G_p^{rand}(s)\}$ digital impulse response and a $\sin(\omega_d t)$ signal, proving that there are 8.485 samples per cycle.

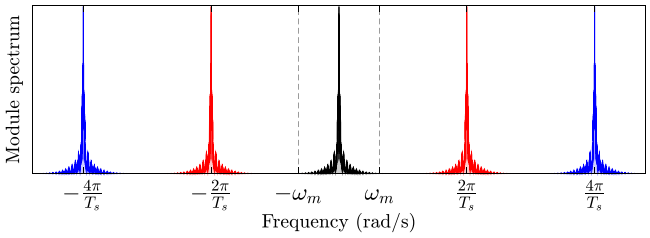

Fig. 15. Amplitude frequency spectrum. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

$$\left.\begin{array}{c} \omega_s \geq 2\omega_m \\ \omega_s = 2\pi/T_s = 125.66 \text{ rad/s} \\ 2\omega_m = 100 \text{ rad/s} \end{array}\right\} 125.66 \geq 100 \tag{16}$$

$$\frac{samples}{cycle} = \frac{2\pi}{T_s\omega_d} \tag{17}$$

$$\begin{array}{c} s^2 + 2\xi\omega_n s + \omega_n^2 = 0 \\ \omega_d = \omega_n\sqrt{1-\xi^2} \end{array} \tag{18}$$

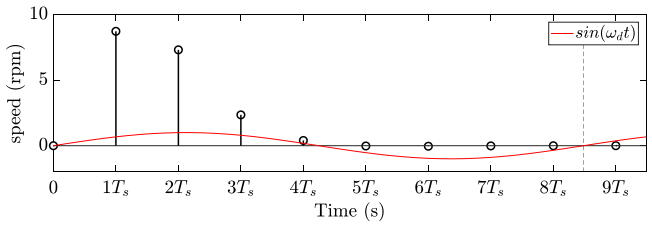

Fig. 16. Sampler period estimation.

3.2. Speed open-loop transfer function, $G^s_{ol}(z)$

Fig. 12 shows that $G^s_{ol}(z)$ is the controller $G_c(z)$ times the Z transform of the product $ZOH(z) \cdot G_p(s)$. The discretization (Eq. (19)) was performed using the ***c2d*** MATLAB function, with the sampler time T_s and the *zoh* option. It could also be manually calculated by means of partial fraction decomposition and z transform basic rules, but it is not the goal of this work.

$$G^s_{ol}(z) = G_c(z) \cdot G_P(z) = G_c(z) \cdot z\left\{\frac{1-e^{-sT_o}}{s} \cdot G_p(s)\right\}$$
$$G^s_{ol}(z) = G_c(z)\,\frac{8.7201(z+0.4215)}{z^2-0.4165z+0.07963} \tag{19}$$

3.3. Position open-loop transfer function, $G^p_{ol}(z)$

Fig. 14 shows that $G^p_{ol}(z)$ is the controller $G_c(z)$ times the Z transform of the product $ZOH(z) \cdot G_p(s) \cdot 6/s$. The factor 6 transforms from rpm to deg/s (Eq. (20)), and the integrator transforms deg/s to deg. The discretization (Eq. (21)) was performed using the ***c2d*** MATLAB function, with the sampler time T_s and the *zoh* option.

$$v\,\frac{\text{rev}}{\text{min}} \cdot \frac{1\text{ min}}{60\text{ s}} \cdot \frac{360\text{ deg}}{1\text{ rev}} = 6v\frac{\text{deg}}{\text{s}} \tag{20}$$

$$G^p_{ol}(z) = G_c(z) \cdot G_P(z) = G_c(z) \cdot z\left\{\frac{1-e^{-sT_o}}{s} \cdot G_p(s) \cdot \frac{6}{s}\right\}$$
$$G^p_{ol}(z) = G_c(z)\,\frac{1.0864(z+2.005)(z+0.1391)}{(z-1)(z^2-0.4165z+0.07963)} \tag{21}$$

4. Controller implementation

In the literature there are many different control strategies, this tutorial aims to use the most common controller (the *PID* and its simplified variants, *P*, *PI*, and *PD*) to set the basics.

In a simple feedback control system, the signal that goes into the controller ($f(t)$) is usually the error between the reference input and the system output: $e(t) = in(t) - out(t)$. Notice that in this work $in(t)$ can be either $pos_r(t)$ or $v_r(t)$ and $out(t)$ can be $pos(t)$ or $v(t)$.

4.1. Controller discretization and coding

There are different methods to discretize a given continuous transfer function. Fig. 17 shows a natural way to understand the derivative discretization, called *Backwards Euler* form. An approximation of $\dot{f}(t)$ between $nT - T$ and nT is the slope of $y(t) = at + b$, and this slope can be calculated for each nT as shown in Eq. (22). Then multiplying both terms by T and doing the z transform with the shift property, Eq. (23) is obtained. Finally, grouping terms, the *Backwards Euler* form is obtained (Eq. (24)).

$$a(nT) = \frac{f(nT) - f(nT-T)}{T} \tag{22}$$

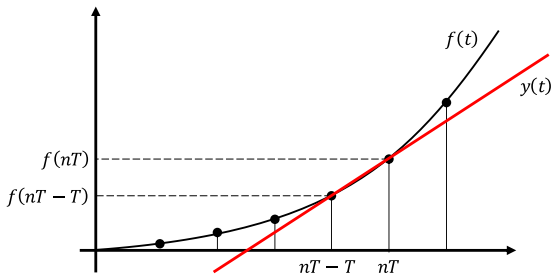

Fig. 17. Backwards Euler derivative graphical interpretation.

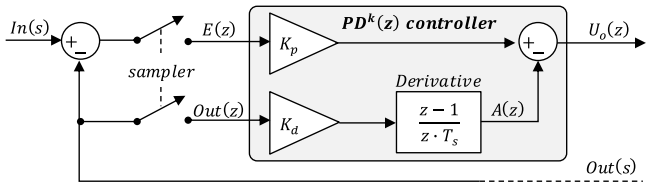

Fig. 18. Anti derivative-kick strategy for a discrete PD controller.

$$A(z)\,T = F(z) - F(z)\,z^{-1} \tag{23}$$

$$D(z) = \frac{A(z)}{F(z)} = \frac{1-z^{-1}}{T} = \frac{z-1}{z \cdot T} \cong s \tag{24}$$

The derivative has two main drawbacks, it amplifies the $f(t)$ high frequency noise and, a sudden jump in $f(t)$ causes $a(t)$ to be instantaneously large (*derivative kick*).

The noise can be filtered by adding a first order pole to $D(z)$ at a desired frequency, N, (Eq. (25)). To only filter the $f(t)$ high frequency noise and, assuming $f(t) = e(t) = in(t) - out(t)$, N should be set above the $out(t)$ highest frequency, which corresponds to the plant natural frequency, ω_d. However, in practice, N must be set according to the nature of $f(t)$ which may have noise at a specific frequency or white noise spread all along the spectrum. For this work the settings $N = 6$ rad/s (speed case) and $N = 50$ rad/s (position case) were used.

$$D^f(z) = D(z)\,\frac{1}{\frac{D(z)}{N}+1} = \frac{z-1}{zT_s} \cdot \frac{1}{\frac{z-1}{zT_s}\frac{1}{N}+1} = \frac{(z-1)\,N}{z\,(NT_s+1)-1} \tag{25}$$

To overcome the derivative kick, the $in(t)$ signal is assumed to be constant (its derivative is zero), then, the derivative is only applied to the $out(t)$ signal, as shown in Eq. (26) and Fig. 18.

$$F(z) = E(z) = IN(z) - OUT(z)$$
$$A(z) = D(z)\,F(z) = D(z)\,(IN(z) - OUT(z)) = -D(z)\,OUT(z) \tag{26}$$

Fig. 19 shows a natural and accurate way to understand the integration discretization, called *Trapezoidal* form. The area under $f(t)$ between $nT - T$ and nT is the sum of the blue rectangle and the red triangle and can be calculated as shown in Eq. (27). The area $y(nT)$ under $f(t)$ between 0 and nT is the current area $A(nT)$ plus all the previous ones (Eq. (28)). Simplifying and doing the z transform with the shift property, Eq. (29) is obtained. Finally, groping terms, the *Trapezoidal* form is obtained (Eq. (30)).

$$A(nT) = Tf(nT-T) + \frac{T\,(f(nT) - f(nT-T))}{2} \tag{27}$$

$$y(nT) = y(nT-T) + A(nT) \tag{28}$$

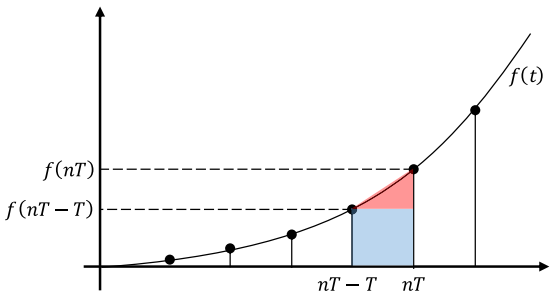

Fig. 19. Trapezoidal integration graphical interpretation.

Table 4
Continuous to discrete controller transformations.

Continuous, $G_c(s) = \frac{U_o(s)}{E(s)}$	Discrete, $G_c(z) = \frac{U_o(z)}{E(z)}$
$P(s) = K_p$	$P(z) = K_p$
$PI(s) = K_p + \frac{K_i}{s}$ $PI(s) = K_p\left(1 + \frac{1}{T_i s}\right)$	$PI(z) = \frac{az+b}{z-1}$ $a = K_p + \frac{K_i T_s}{2} \quad b = \frac{K_i T_s}{2} - K_p$
$PD(s) = K_p + K_d s$ $PD(s) = K_p(1 + T_d s)$	$PD(z) = \frac{az+b}{z}$ $a = \frac{K_d}{T_s} + K_p \quad b = -\frac{K_d}{T_s}$
$PD^f(s) = K_p + K_d \frac{sN}{s+N}$	$PD^f(z) = \frac{az+b}{z+c}$ $a = \frac{K_p + NK_d + NK_p T_s}{NT_s + 1}$ $b = \frac{-(K_p + NK_d)}{NT_s + 1} \quad c = \frac{-1}{NT_s + 1}$
$PID(s) = K_p + \frac{K_i}{s} + K_d s$ $PID(s) = K_p\left(1 + \frac{1}{T_i s} + T_d s\right)$	$PID(z) = \frac{az^2 + bz + c}{z(z-1)}$ $a = Kp + \frac{Ki \cdot T_s}{2} + \frac{Kd}{T_s}$ $b = -Kp + \frac{Ki \cdot T_s}{2} - \frac{2Kd}{T_s} \quad c = \frac{Kd}{T_s}$
$PID^f(s) = K_p + \frac{K_i}{s} + K_d \frac{sN}{s+N}$	$PID^f(z) = \frac{az^2 + bz + c}{z^2 + dz + f}$ $a = \frac{K_p(1 + NT_s) + \frac{K_i T_s}{2}(1 + NT_s) + K_d N}{NT_s + 1}$ $b = \frac{-K_p(NT + 2) + \frac{K_i NT_s^2}{2} - 2NK_d}{NT_s + 1}$ $c = \frac{K_p - \frac{K_i T_s}{2} + NK_d}{NT_s + 1}$ $d = \frac{-NT_s - 2}{NT_s + 1} \quad f = \frac{1}{NT_s + 1}$

$$Y(z) = Y(z)z^{-1} + \frac{T}{2}\left(F(z)z^{-1} + F(z)\right) \tag{29}$$

$$\frac{Y(s)}{F(s)} \xrightarrow{z} \frac{Y(z)}{F(z)} = \frac{T}{2} \cdot \frac{1 + z^{-1}}{1 - z^{-1}} = \frac{T}{2} \cdot \frac{z+1}{z-1} \cong \frac{1}{s} \tag{30}$$

Table 4 shows each controller digital form, which is obtained by replacing the ***s*** complex variable for its ***z*** discrete equivalent expressions (Eqs. (24) and (30)).

In order to program a discrete controller (either using a computer or a microprocessor), its transfer function has to be transformed into an equation in differences by performing the z^{-1} transform. The procedure is explained using the next step by step *PI(z)* example. The transformation of every controller is summarized in Table 6.

Table 5
Anti derivate-kick discrete controllers.

Controller	$U_0(z) = U_0^1(z) - U_0^2(z)$	$U_0^2(z)$
$PD^k(z)$	$P(z)E(z) - K_d D(s)OUT(s)$	$U_0^2(z) = a'\frac{z-1}{z}OUT(z)$
$PID^k(z)$	$PI(z)E(z) - K_d D(s)OUT(s)$	$a' = \frac{K_d}{T_s}$
$PD^{f+k}(z)$	$P(z)E(z) - K_d D^f(z)OUT(s)$	$U_0^2(z) = a'\frac{z-1}{b'z-1}OUT(z)$
$PID^{f+k}(z)$	$PI(z)E(z) - K_d D^f(z)OUT(s)$	$a' = NK_d \; b' = NT_s + 1$

(a) Always keep in mind the controller input and output signals.

$$\frac{U_o(z)}{E(z)} = PI(z)$$

(b) Replace the controller by its expression (Table 4).

$$\frac{U_o(z)}{E(z)} = \frac{az+b}{z-1}$$

(c) Make sure the expression is in terms of z^{-1}

$$\frac{U_o(z)}{E(z)} = \frac{a + bz^{-1}}{1 - z^{-1}}$$

(d) Perform the cross product.

$$U_o(z)\left(1 - z^{-1}\right) = E(z)\left(a + bz^{-1}\right)$$
$$U_o(z) - U_o(z)z^{-1} = aE(z) + bE(z)z^{-1}$$

(e) Solve the equation for the output signal.

$$U_o(z) = aE(z) + bE(z)z^{-1} + U_o(z)z^{-1}$$

(f) Perform the inverse ***z*** transform z^{-1}

$$u_o(kT_s) = a \cdot e(kT_s) + b \cdot e(kT_s - T_s) + u_o(kT_s - T_s)$$

For simplicity

$$u_o(k) = a \cdot e(k) + b \cdot e(k-1) + u_o(k-1)$$

When the anti derivative-kick (*adk*) strategy is applied, the control signal has two terms, $U_0^1(z)$ and $U_0^2(z)$. Table 5 shows the *PD* (z) and the *PID* (z) digital forms using the *adk* strategy (superscript *k*) in combination with the derivative filter (superscript *f*).

4.2. PID tuning

Tuning a *PID* consist of finding the values of the K_p, K_i and K_d which makes the system behave as desired. A more detailed pedagogic report should explain the methods to find those values based on the system response specifications (in time or frequency domain). Unfortunately, it would be too long and it is not the objective of this work. Instead, the MATLAB *PDI Tuner App* is proposed to find a stable and satisfactory step response, with no additional specifications. The MATLAB *PDI Tuner App* needs the open-loop digital transfer function (Eq. (19) or (21), without $G_c(z)$) and the kind of controller to tune (*P*, *PI*, etc.). It is very important to choose the integral and derivative digitalization methods correctly to match the Tables 4 and 6 expressions. The obtained $G_c(z)$ parameters (tuned for step inputs) are shown in Table 7, notice that all K_d values but the *PID* position case, are insignificant, making the derivative hardly noticeable. Table 8 shows the resulting open-loop system Types (number of poles at $z = 1$), which are related to the steady-state error (see Section 5),

Table 6
Controllers' equations in differences.

$G_c(z)$	Equation[a]
$P(z)$	$u_o(k) = K_p \cdot e(k)$
$PI(z)$	$u_o(k) = a \cdot e(k) + b \cdot e(k-1) + u_o(k-1)$
$PD(z)$	$u_o(k) = a \cdot e(k) + b \cdot e(k-1)$
$PD^f(z)$	$u_o(k) = a \cdot e(k) + b \cdot e(k-1) - c \cdot u_0(k-1)$
$PD^k(z)$	$u_o^1(k) = K_p \cdot e(k)$ $u_o^2(k) = a'(out(k) - out(k-1))$ $u_o(k) = u_o^1(k) - u_o^2(k)$
$PD^{f+k}(z)$	$u_o^1(k) = K_p \cdot e(k)$ $u_o^2(k) = \frac{a'}{b'}(out(k) - out(k-1)) + \frac{1}{b'}u_o^2(k-1)$ $u_o(k) = u_o^1(k) - u_o^2(k)$
$PID(z)$	$u_o(k) = a \cdot e(k) + b \cdot e(k-1) + c \cdot e(k-2) + u_o(k-1)$
$PID^f(z)$	$u_o(k) = a \cdot e(k) + b \cdot e(k-1) + c \cdot e(k-2) - d \cdot u_o(k-1) - f \cdot u_0(k-2)$
$PID^k(z)$	$u_o^1(k) = a \cdot e(k) + b \cdot e(k-1) + u_o^1(k-1)$ $u_o^2(k) = a'(out(k) - out(k-1))$ $u_o(k) = u_o^1(k) - u_o^2(k)$
$PID^{f+k}(z)$	$u_o^1(k) = a \cdot e(k) + b \cdot e(k-1) + u_o^1(k-1)$ $u_o^2(k) = \frac{a'}{b'}(out(k) - out(k-1)) + \frac{1}{b'}u_o^2(k-1)$ $u_o(k) = u_o^1(k) - u_o^2(k)$

[a]See Tables 4 and 5 to estimate a, a', b, b', c, d and f variables.

Table 7
Controller's constants.

$G_c(z)$	Speed			Position		
	K_p	K_i	K_d	K_p	K_i	K_d
$P(z)$	0.0530	–	–	0.0571	–	–
$PI(z)$	0.0243	0.3653	–	0.0518	5.19e−3	–
$PD(z)$	0.0497	–	4.6e−5	0.0462	–	2.1e−4
$PID(z)$	0.0371	0.7408	1.1e−4	0.0875	0.0195	0.0054

Figs. 20 and 22 show the closed-loop zero-pole map of each system, where all poles are inside the unit circle (necessary stability condition) and, Figs. 21 and 23 show their respective step responses. Notice that for the PI and PID (position case) there is a pole-zero quasi-cancellation close to z = 1. In practice, those could be canceled having about the same transient response (Fig. 23, red dashed lines) but introducing a step response steady-state error of 1.7% (PI) and 2.4% (PID). For the speed case, the PID step response looks well, while for the position case, PD performs better than the PI and PID. This makes sense because the position PI and PID open-loop systems are Type II (the integral part adds a pole in z = 1) and, the higher the Type, the more difficult to control [32].

4.3. Error estimation

The steady-state error (*sse*), in the case of unity feedback, is defined as the input–output difference once the system reaches the equilibrium ($t = \infty$), in general, Eq. (31). This error depends on the open-loop system type (Table 8) and the input signal.

Table 8
Open-loop system type.

$G_c(z)$	Speed	Position
$P(z)$	0	I
$PI(z)$	I	II
$PD(z)$ and $PD^f(z)$	0	I
$PID(z)$ and $PID^f(z)$	I	II

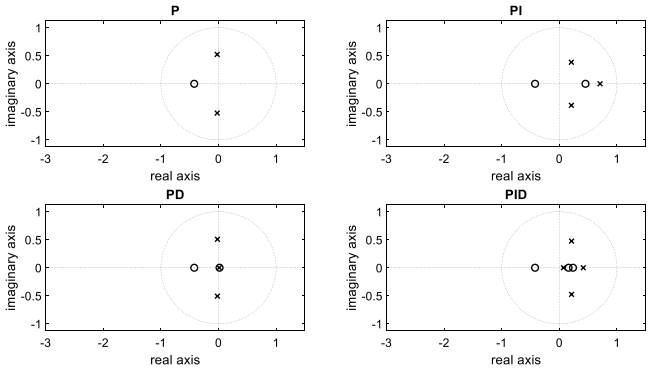

Fig. 20. Speed closed loop zero and pole map representation.

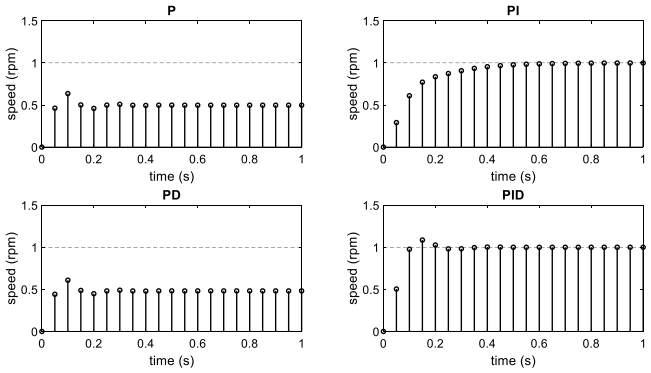

Fig. 21. Speed closed loop steep response.

Table 9
Unit steady-state error calculation.

Input	Open-loop system type		
	0	I	II
Step	$\frac{1}{Kp+1}$	0	0
Ramp	∞	$\frac{1}{Kv}$	0
Acceleration	∞	∞	$\frac{1}{Ka}$

Tables 9 and 10 [27] indicate how to estimate the unit *sse* for the three standard inputs, although the acceleration was not used in this work.

$$sse = \lim_{t\to\infty} e(t) = \lim_{s\to 0}[sE(s)] = \lim_{z\to 1}\left[\left(1 - z^{-1}\right)E(z)\right] \tag{31}$$

Table 10
Static state error.

Open-loop system type	Equation
0	$Kp = \lim_{z\to 1} G_{ol}(z)$
I	$Kv = \lim_{z\to 1} \frac{z-1}{Tz} G_{ol}(z)$
II	$Ka = \lim_{z\to 1} \left(\frac{z-1}{Tz}\right)^2 G_{ol}(z)$

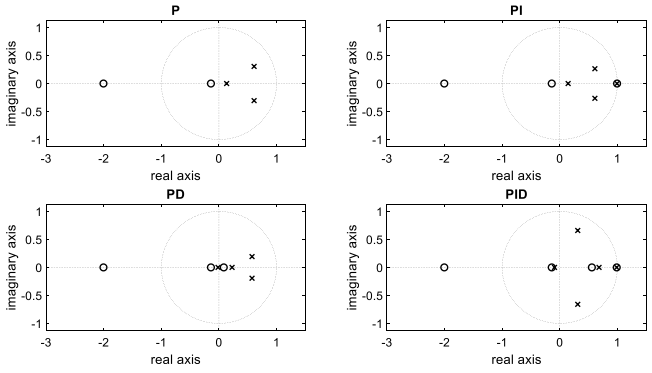

Fig. 22. Position closed-loop zero and pole map representation.

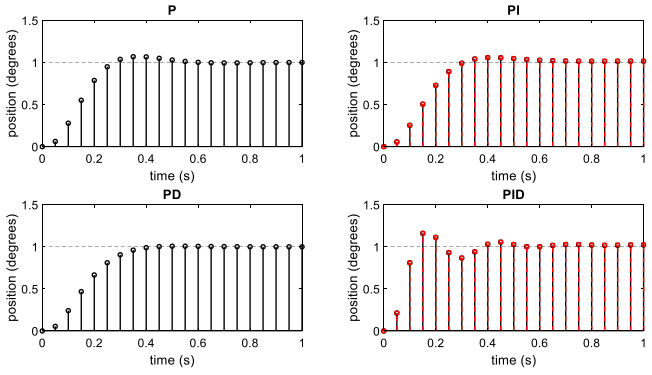

Fig. 23. Position closed-loop steep response. Red dashed lines show the step response when forcing the pole-zero cancellation.. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

5. Nonlinearities implementation

Real systems are rarely LTI (*Linear Time Invariable*). In this simple case, there are three nonlinearities: (a) the motor does not move at low voltages, (b) the motor only supports ±12 V and (c) the integral part may grow excessively.

5.1. Dead zone

The motor is not spinning at low voltages, i.e., for a range between −0.47 V and 0.47 V it does not move. To simulate this non linearity a *Simulink* subsystem was implemented with the following logic (Fig. 24 left) and response (right).

5.2. Saturation

Working under standard conditions the control signal $u_o(k)$ should always remain below the saturation bounds, nevertheless, it is very important to saturate u_o (Figs. 12 and 14) into the

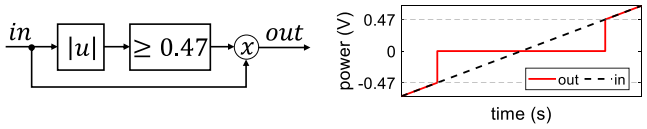

Fig. 24. Dead zone Simulink implementation (left) and its response (right).

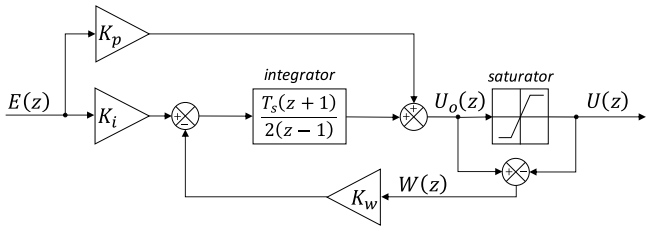

Fig. 25. Anti wind-up block diagram for a PI controller.

Table 11
Anti wind-up controller equation in differences extra term.

$G_c(z)$	Equation[1]
$PI^a(z)$	
$PID^a(z)$	$u_o^a(k) = u_o(k) - \frac{K_w T_s}{2}(w(k) + w(k-1))$
$PID^{a+f}(z)$	
$PID^{a+k}(z)$	$u_o^3(k) = \frac{K_w T_s}{2}(w(k) + w(k-1)) + u_o^3(k-1)$
$PID^{a+f+k}(z)$	$u_o^a(k) = u_o(k) - u_o^3(k)$

plant limits, in this case, ±12 V. In the case of malfunction, the saturation prevents damaging both motor (plant) and controller (actuator). *Simulink* implements a saturation block but it is also necessary to code the saturator in the microprocessor controller function.

5.3. Anti wind-up

The wind-up effect occurs on *PI* and *PID* systems when, for some reason (system malfunction, large set point variation, huge perturbation, etc.), the integral part grows too much and it takes a while to counteract and get back to normal operation. Fig. 25 shows a *PI*(z) controller with an anti wind-up (*awu*) strategy [33]: the saturation difference $w(k) = u_0(k) - u(k)$ and the K_w gain are used to dynamically adjust the integrator signal to prevent it from growing too much. Notice that this method only applies when u_o is saturated. Literature [34] suggests to set K_w according to Eq. (32).

$$K_w^{PI} = \frac{K_i}{K_p} \qquad K_w^{PID} = \frac{1}{\sqrt{K_d/K_i}} \tag{32}$$

The equation in differences for *PI*(z) and *PID*(z) controllers with *awu* are the same shown on Table 6, except for the extra term shown in Table 11.

The *awu* has a peculiar behavior which is very important to have into account. Large values of K_w can cause instability if the *awu* is activated when the system is already saturated, i.e. $w \neq 0$. To illustrate it graphically, Fig. 26 shows 4 different cases, for every case, the PID speed controller, a reference signal of 50 rpm, and a forced power cut off at $t = 1$ s, were used. In cases *a* and *b* the *awu* was applied from the beginning ($t = 0$ s and $w = 0$), as shown, it does not matter if *w* is stable or not, when

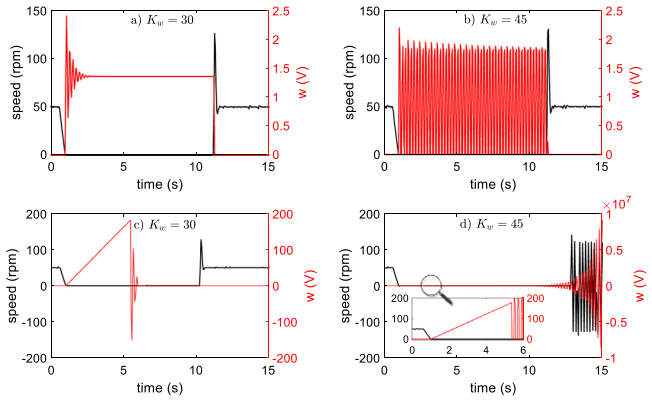

Fig. 26. Anti wind-up initial conditions effect.

the power was restored (t = 11 s) the system went quickly back to normal operation. In case *c* the *awu* was started at $t = 5.4$ s ($w \approx 180$) and the power was restored at t = 11 s, as shown, the system went quickly back to normal. Finally, in case *c* the *awu* was started at $t = 5.4$ s ($w \approx 180$) and the power was restored at t = 12.8 s, as shown, the system became unstable. It is a good practice to select K_w according to Eq. (32) and then verify *w* stability, adjusting K_w if necessary.

Appendix A shows an implementation of the speed $PI(z)$ controller with saturation and *awu* strategy, where (e) is the controller input error, (u0) the controller output, (u) the saturated signal to drive the motor and (w) the saturator difference. The Arduino ADC was set at 12 bits, and the *cPI*() function was called every T_s seconds by a timer interrupt.

6. Experiments and results

This section describes the experiments to validate the real system implementation (code, wiring, and structure) and the theoretical model (Simulink). To do so, four scenarios were considered: (a) How a PID performs with perturbations, (b) The *awu* effect, (c) The derivative drawbacks and (d) The similarity between the real and the simulated system output, for both speed and position cases.

6.1. Perturbation scenario

This test pretends to compare how the designed $PID(z)$ performs in front of a constant and a variable perturbation applied to the motor axis. To do so, a 100 g weight was first attached to the arrow, at *r* cm from the motor axis (Fig. 3, a), creating a variable torque between $\pm 30 \cdot 10^{-3}$ N m (Eq. (33)). Then, the arrow was replaced by a pulley of 6.2 cm in diameter (Fig. 3, b), creating a constant torque of $30 \cdot 10^{-3}$ N m when lifting the weight up at constant speed. With this torque, the dead zone changes; for $-30 \cdot 10^{-3}$ N m the motor starts spinning at 0.3 V (6 rpm) and, for $30 \cdot 10^{-3}$ N m it starts spinning at 2.5 V (31 rpm), giving an asymmetrical dead zone of 2.8 V.

$$M = r \cdot m \cdot g \cdot \sin\alpha$$
$$M_{max} = \left|0.031\ \text{m} \cdot 0.1\ \text{kg} \cdot 9.8\frac{\text{m}}{\text{s}^2} \cdot (\pm 1)\right| \cong |\pm 0.03\ \text{N m}| \quad (33)$$

Eight different speed setpoints were tested, from 25 to 200 rpm. Fig. 27 (variable perturbation) shows the arrow angular speed in two revolutions and its error (computed out of 10 cycles). This error is between 5 and 6% but using a constant perturbation (Fig. 28) this error decreases to around 1%. Theoretically, using a PID, this error should be zero, but in practice, due to some factors, like the low encoder resolution, the reduction gears gaps, etc., it is not perfect.

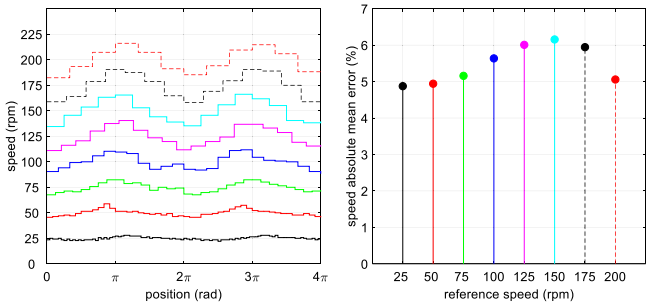

Fig. 27. Motor speed variable perturbation effect.

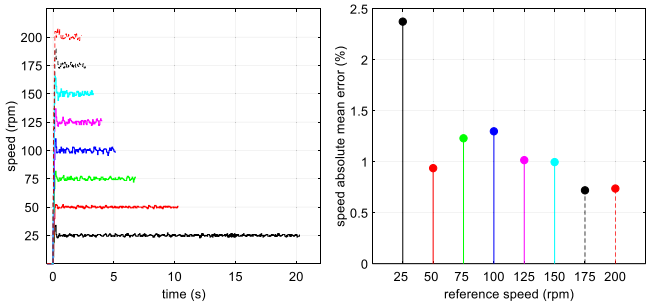

Fig. 28. Motor speed constant perturbation effect. The 100 g weight was pulled up 1.65 m at different speeds.

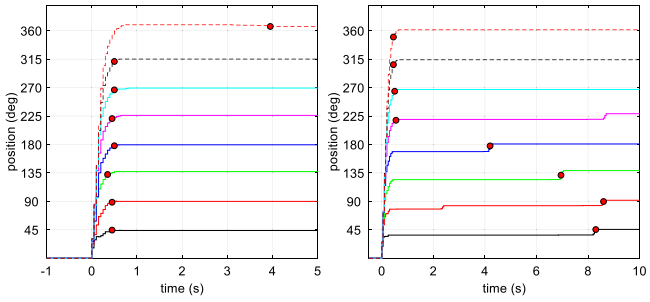

Fig. 29. Motor position: (Left) variable perturbation and (right) constant perturbation.

Also, 8 different position setpoints were tested, from 45 to 360 degrees. Fig. 29 (left) shows the arrow position and how long it takes to reach the desired set point under a variable perturbation (using the 2% settling time criteria). In this case, it takes an average of 0.83 s while using a constant perturbation (Fig. 29, right) it takes an average of 3.75 s. Theoretically, using a PID the position error should be zero, but in practice, because of the large asymmetrical dead zone and the commented motor physical limitations, it is quite improbable to achieve.

6.2. Anti wind-up scenario.

This section shows the speed case system response with and without an *awu* strategy (the position case is omitted as it has equivalent results).

To do the experiments the $PI(z)$ controller, a reference speed of 100 rpm and the K_w (set as default, Eq. (32)) were selected. Fig. 30(a) shows that from 0 to t_1 the system was operating at normal conditions (100 rpm), at t_1 the power supply unit was switch off (simulating a malfunction) and the motor stops (in

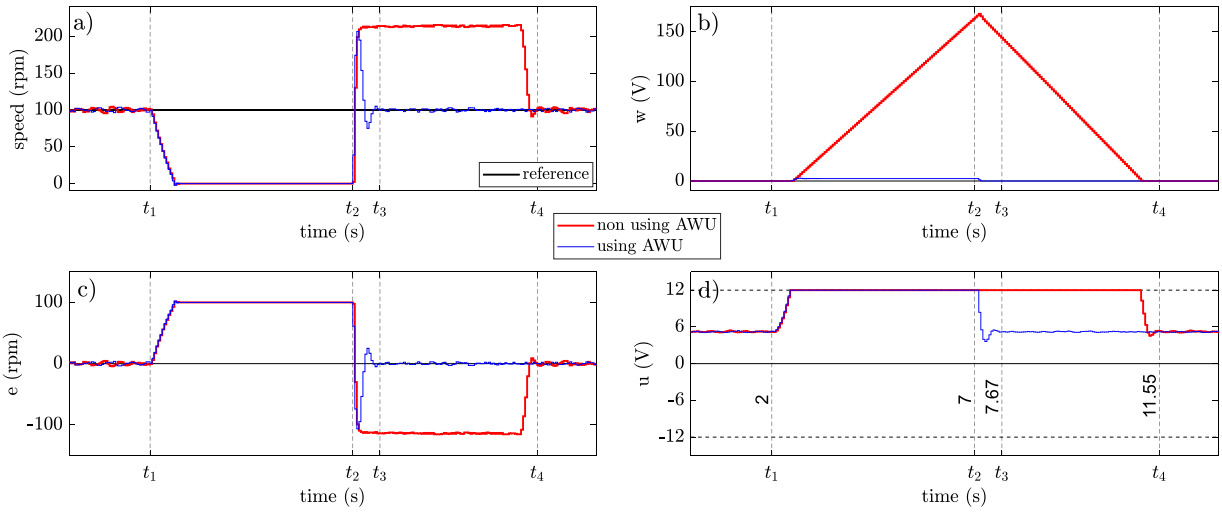

Fig. 30. Anti wind-up performance, where $t_1 = 2$ s, $t_2 = 7$ s, $t_3 = 7.67$ and $t_4 = 11.55$ s.

0.7 s, because of the inertia). At t_2 the power supply unit was switch on and the system got back to normal conditions (100 rpm). Without *awu*, it took 4.55 s (t_4-t_2) while using the *awu* it only took 0.67 s (t_3-t_2).

Without *awu*, w (Fig. 30, b) grew until the power was restored (t_2, 168 V) and then it needs 4.15 s to decrease and get back to zero. Using *awu*, w only grew up to 2.43 V and got back to zero quickly (0.15 s). Fig. 30 also shows the error signal (c), and the control signal (d), which is saturated at 12 V while $w > 0$.

6.3. Derivative filter & anti derivative-kick scenario.

This section compares the system performance applying the derivative filter and the *adk* strategy. To do so, the speed *PID* (z) controller was selected updating its constants to: $K_p = 0.005$, $K_i = 0.7$, $K_d = 0.005$ and $N = 6$. With this K_d value the derivative plays a noticeable role in the speed control system.

Fig. 31 shows the speed response v (rpm) and the control signal u_0 (V) for 5 different cases. The first row shows the *PI* case ($K_d = 0$), were v and u_0 have no significant noise, however, the second row shows the *PID* case, were both v and u_0 have the noise enlarged by the derivative. Also, it shows the derivative kick effect, making $|u_0| > 12$ V when the reference signal suddenly changes ($t = 0$ s and $t = 2$ s). The third row shows the benefits of the derivative filter, were v and u_0 have minimum noise and the v overshoot is clearly smoothed. The fourth row shows the *adk* performance. In this case, u_0 does not grow further than 12 V when the reference signal suddenly changes but it does have the noise amplified. Finally, the fifth row shows the derivative filter in combination with the *adk* strategy. In this case, both v and u_0 are smooth but the system performance is not better than only using the derivative filter.

6.4. Real vs. simulated scenario.

This section compares the real and simulated output to verify the *Simulink* model accuracy (Fig. 32, applying *awu* and derivative filter). The real data (input, output and control signal) was saved into a file. This real input was used to feed the model input. The resulting simulated outputs were then compared with the real ones.

Square (0.25 Hz) and sawtooth (0.5 Hz) waves were used as input references. From −100 to 100 rpm (200 rpm) for the speed test, and from 90° to 270° (180°) for the position test.

Figs. 33 to 48 show the graphical comparison between the real and the simulated data, were:

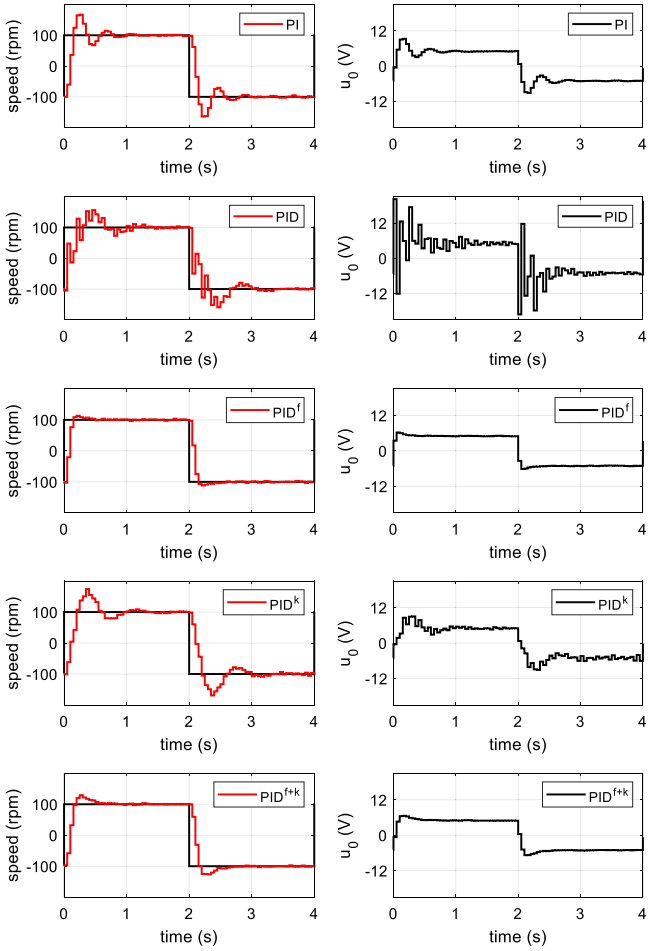

Fig. 31. Speed case *PID* controller derivative influence.

- **Top-left:** Real input (black). Simulated output (blue). Real output (red).
- **Top-right:** Simulated (blue) and real (red) control signal.
- **Bottom-left:** Simulated (blue) and real (red) error.
- **Bottom-right:** Real vs estimated output (black); ideal position (red).

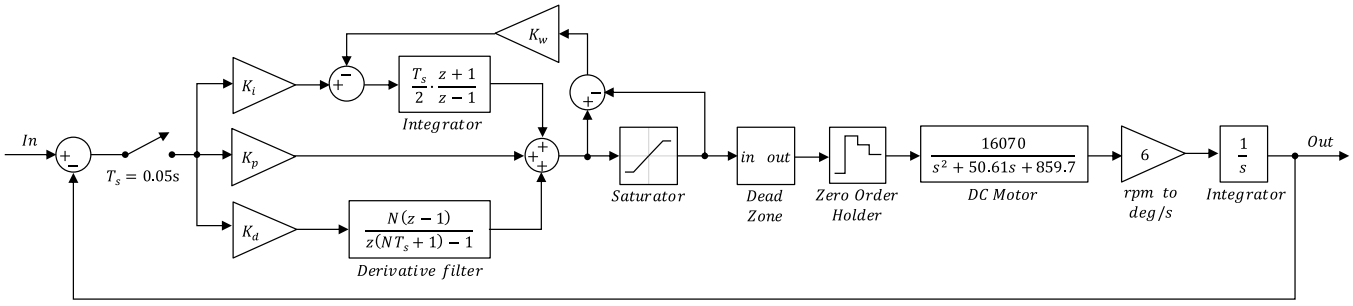

Fig. 32. System block diagram using the *awu* and derivative filter. The real Simulink diagram needs some switches to select between speed, position, P, PI^a, PD^f and PID^{a+f}.

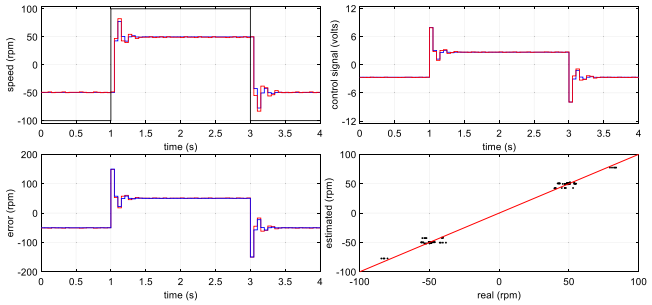

Fig. 33. Speed P controller, step input. (see Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

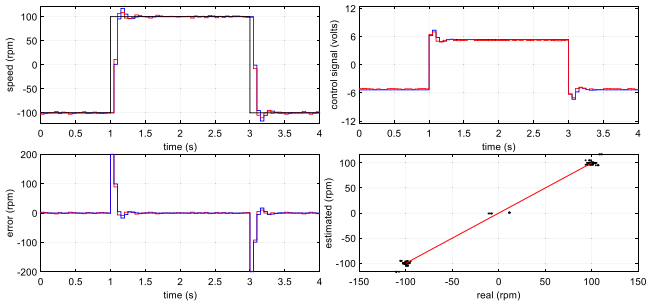

Fig. 36. Speed PID^{a+f} controller, step input. (see Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

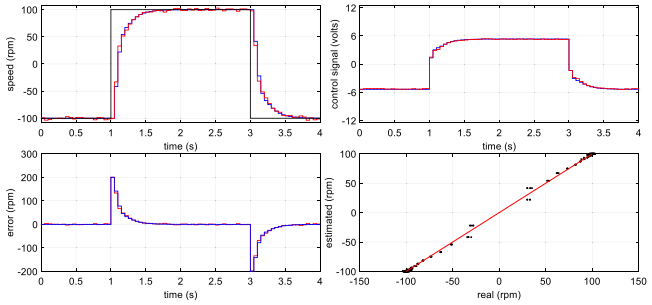

Fig. 34. Speed PI^a controller, step input. (see Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

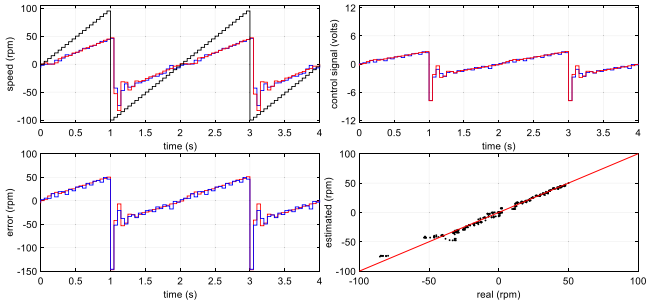

Fig. 37. Speed P controller, ramp input. (see Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

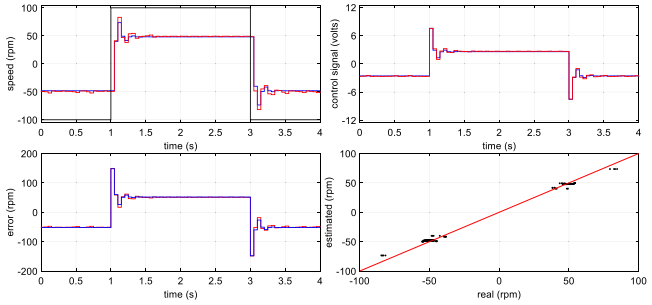

Fig. 35. Speed PD^f controller, step input. (see Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

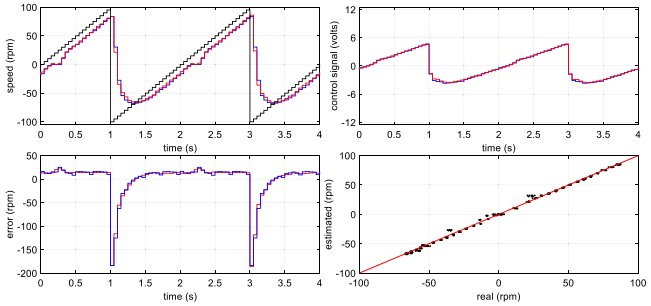

Fig. 38. Speed PI^a controller, ramp input. (see Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 12 shows the numerical comparison between the real and the simulated output using three indicators; the correlation coefficient ρ, the determination coefficient R^2 [35], and the average difference in percentage e.

Table 13 shows the *sse* for every case calculated in three different ways, *Real*, *Simulated* and *Calculated*, the more similar those values are (for each case), the best correlation between real data, simulated data and theoretical results.

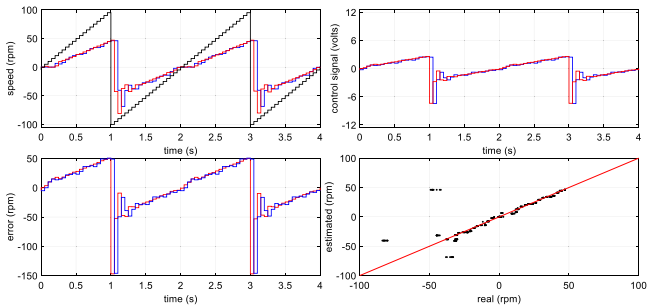

Fig. 39. Speed PD^f controller, ramp input. (see Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

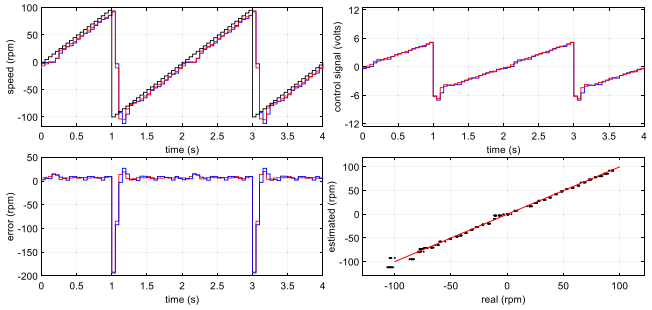

Fig. 40. Speed PID^{a+f} controller, ramp input. (see Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

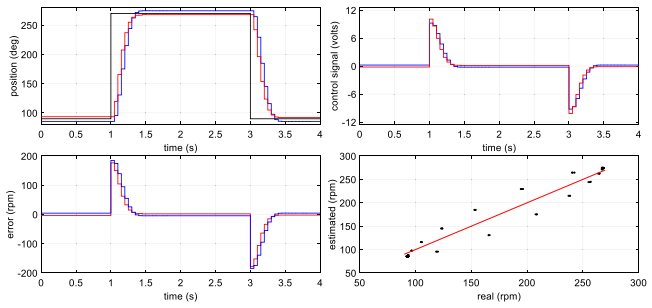

Fig. 41. Position P controller, step input. (see Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

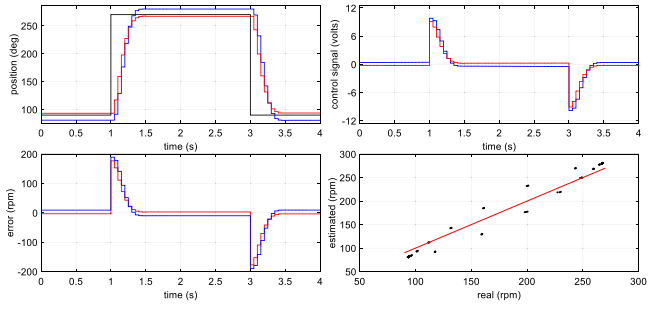

Fig. 42. Position PI^a controller, step input. (see Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

The *sse* has to be calculated according to the inputs, for example, Eqs. (34) and (35) show how to calculate *sse* for the specified ramp.

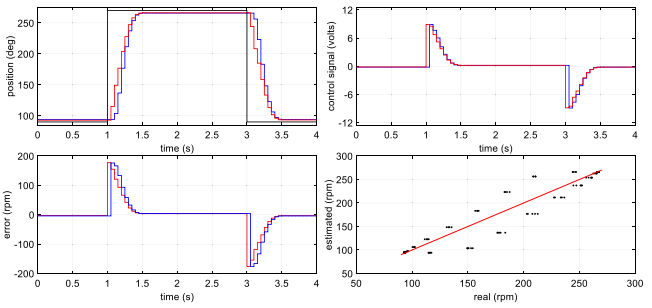

Fig. 43. Position PD^f controller, step input. (see Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

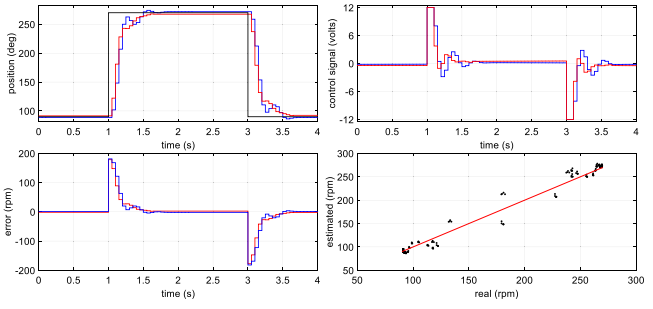

Fig. 44. Position PID^{a+f} controller, step input. (see Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

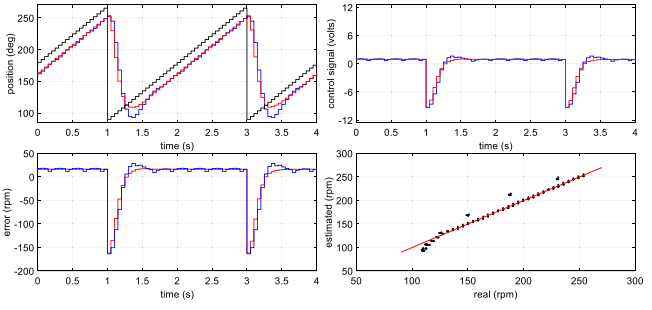

Fig. 45. Position P controller, ramp input. (see Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

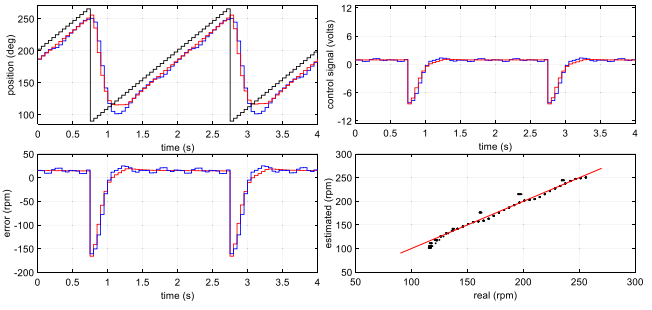

Fig. 46. Position PI^a controller, ramp input. (see Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 12 shows the numerical comparison between the real and the simulated output using three indicators; the correlation coefficient ρ, the determination coefficient R^2 [35], and the average difference in percentage e.

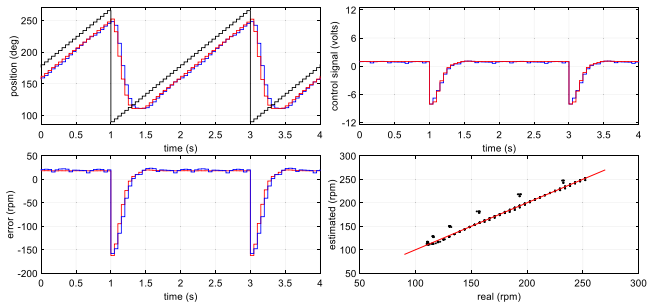

Fig. 47. Position PD^f controller, ramp input. (see Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

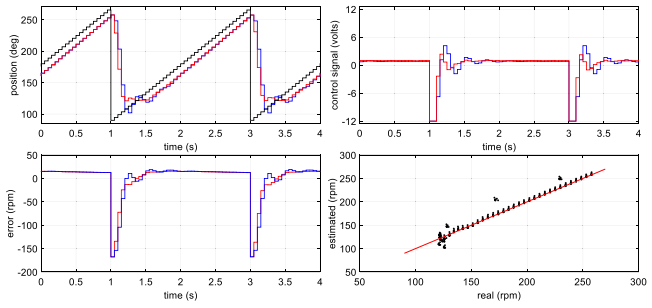

Fig. 48. Position PID^{a+f} controller, ramp input. (Section 6, C for legend). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 13 shows the *sse* for every case calculated in three different ways, *Real*, *Simulated* and *Calculated*, the more similar those values are (for each case), the best correlation between real data, simulated data and theoretical results.

The *sse* has to be calculated according to the inputs, for example, Eq. (34) and (35) show how to calculate *sse* for the specified ramp.

- *Real (R):* Mean difference between the real input and the real output at steady state.
- *Simulated (S):* Mean difference between the real input and the simulated output at steady state.
- *Calculated (C):* Analytically estimated according to the system type and the input signal (Tables 8–10).

$$Speed \qquad sse = \frac{Amplitude}{Period}\frac{1}{Kv} = \frac{200}{\frac{1}{0.5}\cdot Kv} = \frac{100}{Kv} \tag{34}$$

$$Position \qquad sse = \frac{Amplitude}{Period}\frac{1}{Kv} = \frac{180}{\frac{1}{0.5}\cdot Kv} = \frac{90}{Kv} \tag{35}$$

It is important to notice that the highlighted values in Table 13 (worst cases) should be close to zero. They are not because *C* was computed assuming $t = \infty$, and the used signals period was 2 s. To show it graphically, the ramp period was extended up to 80 s (Fig. 49, left). In this case, it is easy to see how the error tends to zero (Fig. 49, right) and, at $t = 2$ s, the error is 15.1°, matching Table 13 ($PI^a(z)$, *Position, Ramp, R*) results (14.9 °). The controllers were tuned to perform well operating with step reference signals, therefore the system performs poorly when operating with ramp reference signals. By adjusting the PI^a controller ($K_p = 0.1$, $K_i = 0.4$ and $K_w = 10$) the error drops to zero in 0.6 s (using the sawtooth reference) but creates an overshoot of 12.6% when using the square reference (Fig. 50).

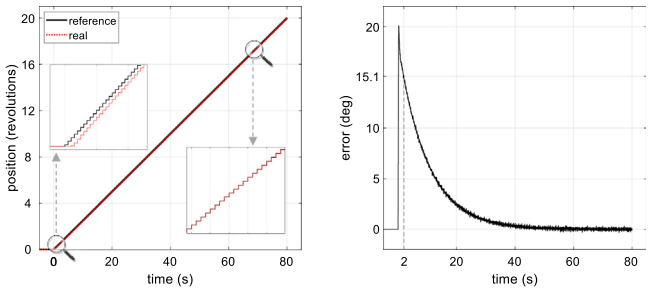

Fig. 49. Steady-state error tendency, PI^a position controller.

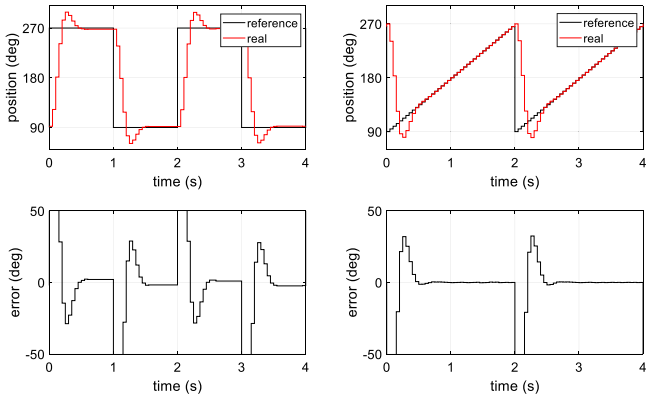

Fig. 50. PI^a controller tuned for ramp reference signals.

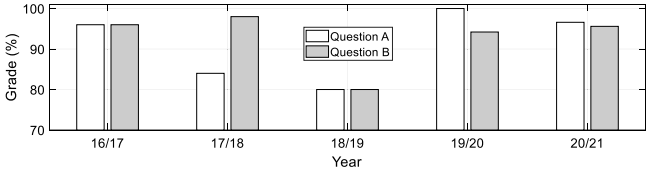

Fig. 51. (A) Was an effective teaching and learning method used? (B) Was the evaluation coherent with the teaching method?

7. Teaching method and results

The proposed teaching platform is used in the subject *System Integration I.* It is a 4th year optional subject in the bachelor's degree in Automation and Industrial Electronic Engineering (University of Lleida, Spain). The goal of this subject is to reinforce the discrete control theory knowledge learned in the *Discrete Process* subject (3rd year). To do so, students have to apply the theory to control the speed and the position of a real DC motor, experiment with different scenarios and compare the real performance with the simulations.

At every class, the professor invests about 15 min teaching or reviewing a specific concept (mostly, summarized in this work) and students have 85 min to practice and ask related questions. Each student is provided a kit which has everything they need to work at home (see Figs. 2 and 3). This kit was especially useful the curse 20/21, allowing students to work 100% at home due the COVID-19 lockdown.

Students invest 35 h (classroom) + 40 h (autonomous work) learning C# (Visual Studio) and Arduino to develop the system interface. The rest of the course (25 h + 50 h) is used to obtain the model transfer function, create and code the controllers, experiment with different scenarios and compare the real system performance with the simulated results.

Table 12
Real vs. estimated output.

$G_c(z)$	Speed						Position					
	Step			Ramp			Step			Ramp		
	ρ	R^2	e	ρ	R^2	e	ρ	R^2	e	ρ	R^2	e
$P(z)$	0.999	0.998	0.65	0.991	0.982	1.34	0.994	0.984	5.05	0.990	0.997	2.20
$PI^a(z)$	1.000	0.999	0.84	0.999	0.997	0.94	0.997	0.978	7.40	0.992	0.980	2.46
$PD^f(z)$	0.999	0.997	0.96	0.823	0.634	3.17	0.989	0.978	2.83	0.989	0.976	1.99
$PID^{a+f}(z)$	0.999	0.999	0.98	0.998	0.996	1.30	0.996	0.99	3.31	0.984	0.96	2.67

Table 13
Steady state error.

$G_c(z)$	Speed (rpm)						Position (degrees)					
	Step			Ramp			Step			Ramp		
	R	S	C	R	S	C	R	S	C	R	S	C
$P(z)$	50.8	50.2	49.9	∞	∞	∞	2.09	4.50	0	16.4	16.2	15.6
$PI^a(z)$	0.05	0.02	0	14.5	14.6	14.4	2.09	11.8	0	**14.9**	**15.3**	0
$PD^f(z)$	50.9	51.8	51.5	∞	∞	∞	3.63	4.09	0	18.0	18.9	19.3
$PID^{a+f}(z)$	4.9e−3	1.9e−7	0	6.71	7.23	7.22	2.53	1.81	0	**14.2**	**14.7**	0

The academic evaluation is divided in four parts, two reports, R1 (12%) and R2 (28%), and two hands-on demonstrations, P1 (18%) and P2 (42%). The R1 and P1 are presented at midterm (evaluating the system interface) and R2 and P2 at the end of the course (evaluating the controllers' implementation). Each part is subdivided in weighted sub-objectives and provided at the beginning of the course (the rubric is known).

This project has been ongoing for the last 5 years with a total of 41 students. At the end of the course students have to fill a subject satisfaction survey. Fig. 51 shows the average grade to the questions: (A) Was an effective teaching and learning method used? and (B) Was the evaluation coherent with the teaching method? In the 19/20 course, students got a didactic adaptation of this document for the first time and the score of question A reached its maximum (100%). There is $\sim$ 18% improvement compared with courses 17/18 and 18/19 but only 4% compared with the first year (16/17). This may be because the first year (due to inexperience) the professor invested extra time providing individual assistance and guidance. In the 20/21 course, question A scored 96.6%, which is remarkable considering that the classes where all virtual due the COVID-19 lockdown. Besides this indicators, using this guide, students accomplished the main objectives earlier and could experiment with other control strategies beyond the rubrics requirements.

Notice that the student's platform uses a different motor (BS138F-2S-6-21) to force students to do all the steps by themselves.

8. Conclusions

This paper describes the materials and methods used to implement a didactic platform designed to teach or reinforce discrete control theory concepts to undergraduate students.

The platform is based on a 3D printed framework (5€), a 12 V DC motor (42€), an Arduino Due (28€) and a motor driver (30€), with a total cost of 105€ per unit.

Students, from their knowledge, the information summarized in this document, and the professor advises, had to:

(a) Create a system interface using C#.
(b) Implement the C code to get the motor speed and position.
(c) Adjust the motor sample time at 50 ms, which theoretically produces a speed error of $\pm$10% at 7 rpm and $\pm$0.1% at 220 rpm.
(d) Obtain the motor transfer function model by analyzing its differential equations and calculate its values using a set of random steps and the MATLAB *System Identification ToolBox*. As a result, the obtained transfer function had a fitting of 95.57%
(e) Adjust the system sampling period at 50 ms. To do so, the Fourier transform and Shannon's theorem were used.
(f) Draw the whole system model to do the math properly.
(g) Do the continuous to discrete plant transformation. To do it quickly the **c2d** MATLAB function was used, but it could be easily done manually.
(h) Design the P, PI, PD, and PID digital controllers. The chosen digitalization method was the *Trapezoidal* form for the integrator and the *Backwards Euler* for the derivative.
(i) Transform the designed controllers into their equations in differences and write the code in the microprocessor.
(j) Implement the nonlinearities, both in the microcontroller and the Simulink model.
(k) Theoretically calculate the stationary sate errors.
(l) Perform a set of experiments to see the system behavior in different scenarios.

Four different scenarios were tested. The first one using a 100 g weight as a perturbation. In the speed case, results show that the PID controller keeps the error around 1% if the perturbation is constant but this error increases (up to 6%) in front of a variable perturbation. In the position case either using a constant or a variable perturbation, the error tends to zero.

Fig. 29 (right) shows that the settling time decreases when the reference position increases. It may seem counterintuitive but it makes sense thinking in terms of dead zone and inertia. On the one hand, large reference steps (225° and above) create big errors and strong control signals, achieving the reference position with

inertia in approximately 0.5 s. On the other hand, small steps (180° and below) create small errors and weak control signals. After the initial motor rotation, with almost no inertia, the error signal is too small and the control signal falls into the dead zone, consequently, the motor is not moving until the integral part grows enough to pass the dead zone, taking between 4 and 9 s. This phenomenon can also be observed in Fig. 41 and indicates that the *Simulink* model is accurate enough as the simulated position follows the same pattern.

The second scenario tests the implemented *awu* by forcing a system malfunction (shutting the plant power down). Results show that the used *awu* strategy provides a fast recovery. Notice that in case of malfunction or large set point variation the *awu* only plays a role in the integral part. To deal with the derivative kick it is recommended to feed the derivative part not with the error signal but with the output signal, in this case, the motor speed or position. This is discussed in the third scenario (Fig. 31), comparing the effect of applying a derivative filter in combination with an *adk* strategy. Results show that a derivative filter with a proper N value is enough to overcome the derivative-kick and to obtain a smooth control signal.

The fourth scenario (Figs. 33 to 48) compares the real data (in red) and the simulated (in blue) to verify the *Simulink* model accuracy. The real input was stored and used as the simulated input. Figs. 44 and 48 show the two single cases were the control signal saturates and the *awu* reacts. In order to evaluate simulation accuracy, Table 12 shows the numerical comparison between the real and the simulated output, showing an average $\rho = 0.983$, $R^2 = 0.965$, and $e = 2.38\%$. Also, Table 13 compares the real, the simulated and the calculated steady-state error, showing that, for every case, they are very similar, which means the theory matches the model and the real world. This information is key for students to trust simulators and to realize that the theoretical concepts, learned in theory classes, apply to real-world problems.

In terms of performance, the controllers behave poorly when using ramp reference signals, for example, Fig. 46 (PI^a) shows an error of 14.9° (at $t = 2n - 1|\mathbb{N}^+$) and needs 80 s to eliminate this error (Fig. 49). Tuning the PI^a controller for ramp inputs reduces this time from 80 to 0.6 s (Fig. 50) but increases the overshoot from 0 to 12.6% (step inputs).

As a result, it is important to tune the controller for the expected reference signals or implement an advanced method to dynamically adjust the controller to provide the desired response. This paper uses the MATLAB PID tuner App to adjust the P, PI, PD, and PID controllers and future work will show how to implement an adaptive PID tuner to adapt changes in system dynamics.

From an academic point of view, the proposed low cost teaching platform allows students to put in practice discrete control theoretical concepts and learn from real issues encountered while doing the project. The platform is suitable for teaching basic controllers in bachelor's degree as well as advanced techniques in master's degree. Results show that using a complete manual like this one helps professors guide the students' learning process effectively.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A

Simplified Arduino code to implement a PI controller. Notice the loop function is empty and all functions are called by external (encoder biphasic signal) or internal (timer) system interruptions.

```
#include<DueTimer.h>
const byte PinA = 2;      const byte PinB = 3;
const byte PinPWM = 11;   const byte PinDir = 30;
volatile bool signalA = false;
volatile bool signalB = false;
volatile long int np =0;
volatile long int npAnt =0;
float position = 0;
float speed = 0;
float Ts = 0.05;
float e,eAnt,w,wAnt,u,u0,u0Ant

void setup(){
analogReadResolution(12);
pinMode(PinA,INPUT_PULLUP);
pinMode(PinB,INPUT_PULLUP);

attachInterrupt(digitalPinToInterrupt(PinA),funA,
CHANGE);
attachInterrupt(digitalPinToInterrupt(PinB),funB,
CHANGE);

Timer1.attachInterrupt(funTimer).setPeriod(Ts*1e6)
.start();
}

void loop() {}

void funA(){
signalA=digitalRead(PinA)==HIGH;
if(signalA!=signalB) np++;
else np--;
}

void funB(){
signalB=digitalRead(PinB)==HIGH;
if(signalA==signalB) np++;
else np--;
}

void funTimer(){
   position = (float)np*0.2205;
   speed = 0.2205 * (float)(np - npAnt)/T1;
   npAnt = np;
   cPI(); // call the controller function
}

void cPI()
{
  e = getReferenceSpeed() - speed;
  u0 = (2*kp + ki*Ts)/2 * e +
       (ki*Ts - 2*kp)/2 * eAnt +
        u0Ant -
        kw*Ts/2 * (w + wAnt); //Anti wind-up
  u0Ant = u0;
  eAnt = e;
  wAnt = w;

  u = u0; //saturation
  if(u0 > 12 ) u = 12;
  if(u0 <-12 ) u= -12;
  w = u0 - u;

  if (u < 0)  digitalWrite(dir,HIGH);
  else        digitalWrite(dir,LOW);

  analogWrite(PinPWM,round(abs(u)*(4095/12)));
}
```

Appendix B

Simplified MATLAB code to plot the module spectrum of a signal using the fast Fourier transform.

```
s = tf('s');
Ts = 0.05; %sampling period (s)
To = 60; %signal period (s)
t = 0:0.01:To; %time vector (s)
G = 16070 / (s^2 + 50.61*s + 859.7); %transfer function
N = numel(t); %number of elements of the t vector
fo = 2*pi/To; %signal frequency (rad/s)
ws = 2*pi/Ts; %sampling frequency (rad/s)
in = createRandSteps(To,2); %cerates input signal
simOut = lsim(G,in,t); %output signal simulation
w = [(0:fo:fo*(N-1)) - fo*(N-1)/2]; %frequency vector
m = abs(fftshift(ifft(simOut))); %inv. fast F. trans.
plot(w,m,'k');
plot(w+ws,m,'r'); %1st positive compl. component
plot(w+2*ws,m,'b'); %2nd positive compl. component
```

References

[1] Bird J. Higher engineering mathematics. London: Routledge; 2014.
[2] Astrom KJ, Lundh M. Lund control program combines theory with hands-on experience. IEEE Control Syst Mag 1992;12(3):22–30.
[3] Apkarian J, Åström KJ. A laptop servo for control education. IEEE Control Syst 2004;24(5):70–3.
[4] https://www.sidilab.com/productos/control-difuso-carro-con-pendulo-invertido (Accessed on 15/03/2020).
[5] Ramasamy S, Pradhan HV, Ramanathan P, Arulmozhivarman P, Tatavarti R. A novel and pedagogical approach to teach PID controller with LabVIEW signal express. In: 2012 IEEE International Conference on Engineering Education: Innovative Practices and Future Trends (AICERA), Kottayam. 2012; p. 1–8.
[6] Ayasun S, Nwankpa CO. Induction motor tests using MATLAB/Simulink and their integration into undergraduate electric machinery courses. IEEE Trans Educ 2005;48(1):37–46.
[7] Goodwin GC, Medioli AM, Sher W, Vlacic L, Welsh JS. Emulation-based virtual laboratories: A low-cost alternative to physical experiments in control engineering education. IEEE Trans Educ 2011;54(1):48–55.
[8] https://es.mathworks.com/products/simulink.html (Accessed on 15/03/2020).
[9] https://www.ni.com/es-es/shop/labview.html (Accessed on 15/03/2020).
[10] Ionescu CM, Fabregas E, Cristescu SM, Dormido S, De Keyser R. A remote laboratory as an innovative educational tool for practicing control engineering concepts. IEEE Trans Educ 2013;56(4):436–42.
[11] He Zhenlei, Shen Zhangbiao, Zhu Shanan. Design and implementation of an internet-based electrical engineering laboratory. ISA Trans 2014;53(5):1377–82.
[12] Soriano A, Marín L, Vallés M, Valera A, Albertos P. Low cost platform for automatic control education based on open hardware. IFAC Proc 2014;47(3):9044–50.
[13] Vlachos C, Williams D, Gomm JB. Solution to the Shell standard control problem using genetically tuned. Control Eng Pract 2002;10(2):151–63.
[14] Liu GP, Daley S. Optimal-tuning PID. Control for industrial systems. Control Eng Pract 2001;9(1):1185–94.
[15] Temel S, Yağli S, Gören S. P, pd, pi, pid controllers. Middle East Technical University, Electrical and Electronics Engineering Department; 2013.
[16] Ziegler JG, Nichols NB. Optimum settings for automatic controllers. Trans ASME 1942;64:759–68.
[17] Papadopoulos Konstantinos G, Yadav Praveen K, Margaris Nikolaos I. Explicit analytical tuning rules for digital PID controllers via the magnitude optimum criterion. ISA Trans 2017;70:357–77.
[18] Tan Wen, Liu Jizhen, Chen Tongwen, Marquez Horacio J. Comparison of some well-known PID tuning formulas. Comput Chem Eng 2006;30(9):1416–23.
[19] Precup R, Preitl S, Radac M, Petriu EM, Dragos C, Tar JK. Experiment-based teaching in advanced control engineering. IEEE Trans Educ 2011;54(3):345–55.
[20] Khan S, Jaffery MH, Hanif A, Asif MR. Teaching tool for a control systems laboratory using a quadrotor as a plant in MATLAB. IEEE Trans Educ 2017;60(4):249–56.
[21] Enikov ET, Campa G. Mechatronic aeropendulum: Demonstration of linear and nonlinear feedback control principles with MATLAB/Simulink real-time windows target. IEEE Trans Educ 2012;55(4):538–45.
[22] El-Shafei MAK, El-Hawwary MI, Emara HM. Implementation of fractional-order PID controller in an industrial distributed control system. In: 2017 14th international multi-conference on systems, signals & devices (SSD). 2017, p. 713–8.
[23] Gunasekaran M, Potluri R. Low-cost undergraduate control systems experiments using microcontroller-based control of a DC motor. IEEE Trans Educ 2012;55(4):508–16.
[24] Aldeyturriaga Ricardo OG, Junior Carlos AAL, Silveira Antonio S, Coelho Antonio AR. Low cost setup to support PID ideas in control engineering education. IFAC Proc 46(17):19–24.
[25] https://es.mathworks.com/help/control/ref/pidtuner-app.html (Accessed on 15/03/2020).
[26] https://www.arduino.cc/en/Guide/ArduinoDue (Accessed on 15/03/2020).
[27] https://www.pololu.com/product/1213 (Accessed on 15/03/2020).
[28] https://www.pololu.com/product/3240/specs (Accessed on 15/03/2020).
[29] Golnaraghi F, Kuo B. Automatic control systems. 9th ed.. John Wiley & Sons; 2009.
[30] Shannon Claude. Communication in the presence of noise. Proc IRE 1949;37(1):10–21.
[31] Ogata Katsuhiko. Discrete-time control systems. 2nd ed.. Prentice-Hall International; 1995.
[32] Ogata Katsuhiko. Modern control engineering. 5th ed.. Pearson; 2010.
[33] Aström Karl J, Murray Richard M. Feedback systems, an introduction for scientists and engineers. Princeton University Press; 2019.
[34] Aström Karl J, Hägglund Tore. ID controllers: Theory, design, and tuning. 2nd ed. The Instrumentation, Systems, and Automation Society.
[35] Fisher RA. Statistical methods for research workers. 13th ed.. Hafner; 1958.